Decoding

and Decrypting Network Packets with Wireshark

In the article I will cover dissecting and decrypting Bluetooth High Speed over wireless traffic.

The main idea is that well known Bluetooth protocols, profiles and security mechanisms to be used with secondary radio are already present in many devices. Given that secondary radio is usually significantly faster we achieve faster data transfer while keeping existing API. The user does not need to wory about changing his code. See [1] for more details.

There are two flows of traffic during High Speed data transfers. One is coming through BR/EDR Bluetooth channel and the other through a wireless 802.11 interface. In this article decoding wireless traffic will be covered. Since an L2CAP connection is established through Bluetooth, the wireless dump lacks the connection signalling packets and therefore Wireshark cannot find out which protocol is in use on upper layers. Wireshark also needs Bluetooth the key to be able to decrypt wireless frames.

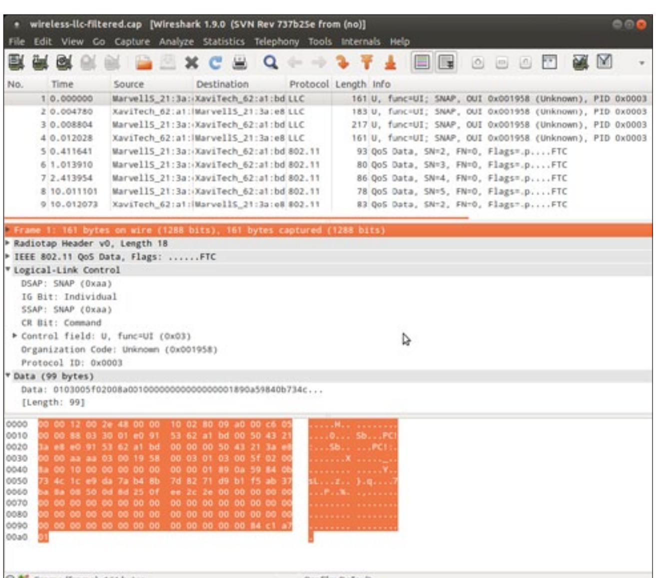

Figure 1. *Captured wireless traffic*

Encryption Basics

Connections between High Speed devices are encrypted and share symmetric keys. In 802.11 it has name Pairwise Transient Key. The PTK is generated by concatenating the following attributes: PMK, AP nonce (ANonce), STA nonce (SNonce),

Listing 1. *Registration of Bluetooth OUI*

```
#define OUI_BLUETOOTH   0x001958 /* Bluetooth
                  SIG */

void proto_register_bt_oui(void)
{
    static hf_register_info hf[] = {
       { &hf_llc_bluetooth_pid,
          { "PID",    "llc.bluetooth_pid",
                 FT_UINT16, BASE_HEX,
              VALS(bluetooth_pid_vals), 0x0,
                 "Protocol ID", HFILL }
       }
    };

    llc_add_oui(OUI_BLUETOOTH, "llc.bluetooth_
                 pid", "Bluetooth OUI PID",
                 hf);
}
```

Listing 2. *Types of Bluetooth High Speed frames*

```
#define AMP_U_L2CAP      0x0001
#define AMP_C_ACTIVITY_REPORT 0x0002
#define AMP_C_SECURITY_FRAME   0x0003
#define AMP_C_LINK_SUP_REQUEST 0x0004
#define AMP_C_LINK_SUP_REPLY   0x0005

static const value_string bluetooth_pid_vals[] = {
   { AMP_U_L2CAP,        "AMP_U L2CAP ACL data" },
   { AMP_C_ACTIVITY_REPORT,  "AMP-C Activity Report" },
   { AMP_C_SECURITY_FRAME, "AMP-C Security frames" },
   { AMP_C_LINK_SUP_REQUEST,  "AMP-C Link supervision request" },
   { AMP_C_LINK_SUP_REPLY, "AMP-C Link supervision reply" },
   { 0,  NULL }
};
```

Listing 3. *Registering eapol and btl2cap dissectors*

```
void proto_reg_handoff_bt_oui(void)
{
   dissector_handle_t eapol_handle;
   dissector_handle_t btl2cap_handle;

   eapol_handle = find_dissector("eapol");
   btl2cap_handle = find_dissector("btl2cap");

   dissector_add_uint("llc.bluetooth_pid", AMP_C_SECURITY_FRAME, eapol_handle);
   dissector_add_uint("llc.bluetooth_pid", AMP_U_L2CAP, btl2cap_handle);
}
```

Listing 4. *Adding second LLC header*

```
file: epan/crypt/airpdcap.c function: AirPDcapPacketProcess

const guint8 bt_dot1x_header[] = {

        0xAA,             /* SSAP=SNAP */
        0x03,             /* Control field=Unnumbered frame */
        0x00, 0x19, 0x58, /* Org. code=Bluetooth SIG */
        0x00, 0x03        /* Type: Bluetooth Security */
};

/* Filter 802.1X authentication frames */
if (memcmp(data+offset, dot1x_header, 8) == 0 ||
    memcmp(data+offset, bt_dot1x_header, 8) == 0) {
```

AP MAC address, and STA MAC address. Terminology 802.11 means: STA – station and AP – access point, for High Speed initiator and responder, a nonce is an arbitrary number used only once in a cryptographic communication. PMK is a shared secret key between two AMP controllers. It is valid throughout the whole session and needs to be exposed as little as possible. For more information see [3].

Getting Pairwise Master Key (PMK)

Bluetooth provides key material for wireless security by creating Dedicated AMP Link Key which is used by wireless devices as Pairwise Master Key. The PMK is needed for decrypting wireless encrypted frames.

After we pair two devices (SSP pairing is needed) bluetooth creates Bluetooth Link Keys (LK) which are usually stored. In Linux, the LK can be found in the following path:

```
/var/lib/bluetooth/<MAC Address>/linkkeys .
```

First we create Generic AMP Link Key (GAMP) given known LK.

`GAMP _ LK = HMAC-SHA-256(LK||LK, 'gamp', 32)` where `LK||LK` means concatenations of 2 16 bits Link Keys forming 32 bit result array. Then we create Dedicated AMP Link Key.

`Dedicated _ AMP _ Link _ Key=HMAC-SHA-256(GAMP _ LK, '802b', 32)`. See [2] "Vol 2: 7.7.5 The Simple Pairing AMP Key Derivation Function h2" for more info.

The result PMK will be used by wireshark decryption engine after some modification below.

Decoding Bluetooth High Speed Traffic Over Wireless

Figure 1 shows captured wireless traffic taken with an external wireless card in monitor mode filtered by MAC addresses. We see two types of frames: LLC frames and 802.11 data which Wireshark was able to decode. Since we know that all High Speed frames shall have LLC headers we might assume that those frames without LLC headers are encrypted and that means that authentication and key generation is happening in packets marked as LLC.

The Bluetooth specification specifies encapsulation methods used for data traffic in [2] "Vol 5: Table 5.1: 802.11 AMP LLC/SNAP encapsulation." Wireshark already has LLC dissector and we only need to define our *Organization Unique Identifier* (OUI) or Company Id and then register our OUI like it is shown in Listing 1.

Once complete, packets with Bluetooth OUI will be identified as Bluetooth High Speed packets. The field `llc.bluetooth _ pid` identifies the type of data the packet contains. Listing 2 shows all possible data types.

What we have now is only LLC is dissected. The data coming after LLC header is dissected as raw data. We want Wireshark to dissect encapsulated frames from Wireshark's known protocols list since the tool already has almost all major protocol supported. For that we need to register dissectors of known protocols according to their bluetooth_pid values to LLC dissector table. AMP Security frames represents X11 Authentication which might be decoded by eapol dissector, AMP L2CAP ACL data frames might be decoded by btl2cap dissector.

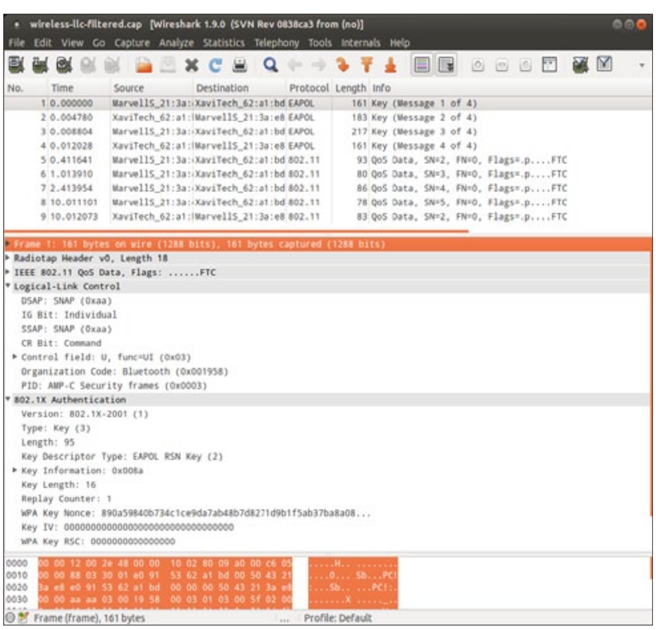

Figure 2. *Decoding EAPOL packets*

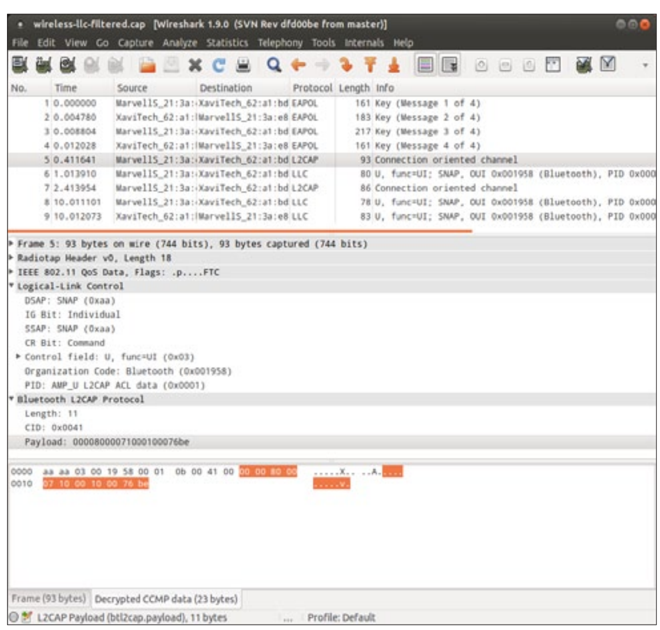

Figure 3. *Decoding L2CAP packets in decrypted CCMP data*

References

- [1] Bluetooth High Speed. *http://www.bluetooth.com/Pages/High-Speed.aspx*
- [2] BLUETOOTH SPECIFICATION Version 4.0 *https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737*
- [3] IEEE 802.11i-2004: Amendment 6: Medium Access Control (MAC) Security Enhancements *http://standards.ieee.org/getieee802/download/802.11i-2004.pdf*

Listing 3 shows adding L2CAP and EAPOL dissectors in the dissector table. First we find dissector handles with `find _ dissector` and then we add handles with `dissector _ add _ uint`.

The change above allows Wireshark to decode EAPOL frames from the dump. Figure 2 shows Wireshark dissecting EAPOL frame, the first message in the 4-way authentication sequence.

After the EAPOL frames traffic is encrypted. This is because the authentication LLC header is also encrypted and those packets cannot be identified as Bluetooth High Speed data. We need to decrypt the packets and then Wireshark is able to understand the packet by looking at the decrypted LLC.

Decrypting Bluetooth Encrypted Data

Next step is to determine the decryption key. Fortunately we have all the required information like Bluetooth supplied PMK and trace containing the 4-way authentication. Wireshark already has the capability to derive *Pairwise Transient Key* (PTK) from a 4-way authentication sequence (shown as EAPOL in Wireshark) in the airpdcap library.

Bluetooth EAPOL frames are not recognized because airpdcap tries to only decode packets with special LLC header specifying type `0x88, 0x8E /* Type: 802.1X authentication */`. The solution is to add second LLC header and filter only those two headers shown in Listing 4.

After this change airpdcap is able to find PTK key (given that PMK key is known by Wireshark through preferences) and then decrypt data traffic. Figure 3 shows.

ANDREI EMELTCHENKO

Author has over 12 years of experience working with network protocols in Nokia, Nokia Siemens Networks and Intel.